## The Assignment

We have to implement a Linux device driver that provides two devices where user processes can read and write data. The confusing part is that data written to one device actually ends up in the opposite device. As with the previous assignment, we have been given a lot of the code in advance, we just have to put it together and think about it afterwards.

## The Implementation

### The Device Driver (listing 1)

My implementation builds heavily on the `scull/pipe.c` code. The advanced functions such as polling and asynchronous access were removed, and the remaining functions were renamed to better fit DM14. I've left most of the original comments intact. The conversion was pretty much just removing all the extras that I didn't need.

The biggest change to the scull code is in `dm14_open()` and `dm14_release()`. The first has been modified to affect both buffers, and the second is cleaned out…there isn't anything to release, unless the whole driver is unloaded. The scull code also kept track of the number of readers and writers in order to dynamically allocate the buffers when needed, but I wanted the data to be persistent.

The read function is pretty much the original, and the write function likewise with the exception of accessing the opposite device (as `(num+1)%2`).

There is extensive use of semaphore locking, wait queues, and such bells and whistles, so it should work without issues on SMP systems.

With the added char device registration on init, that is really all it takes to modify the scull code to fit the assignment. Can hardly take credit for it…but no need to reinvent the wheel.

### The Test Program (listing 2)

Testing the device driver is more interesting, but it is limited what can be tested. Initializing and removing the driver will nearly always succeed, as will opening and closing the devices. And as we saw in the last assignment about region memory, `copy_to/from_user()` is quite safe and will catch invalid addresses.

Here I also built on the given `testprog.c`, but I am using the standard `fopen/fread/fwrite()` functions instead of `open/read/write()`. This is beneficial since it will handle buffering and partial writes internally, plus collating many small writes into one large. Which in turn means there are less potential errors to check for.

More detail about how the test program works follows in the next section.

**The Testing Phase**

The first phase tests the non-blocking I/O. This is done in a while loop that makes sure each of the combinations dev0:read, dev0:write, dev1:read, and dev1:write happens 1000 times. This simulates a producer/consumer relationship and might as well be between two or more processes, or across a network via sockets.

The first attempt to read data results in an `EWOULDBLOCK` error, since there is no data available yet. The subsequent calls all succeed on first try, since this is a single processor system and a single thread, so it all happens sequentially. But the first read proves that non-blocking I/O does work (as it should, since it's the scull code).

The second phase is in regular blocking mode, and forks in order to test the semaphore locking. Parent process opens dm14-0, and forked process opens dm14-1, whereafter they cycle through a producer/consumer relationship. The input data is summed up and compared with the opposite device's output data, and if they don't match there is a problem.

If one of the processes were to die or be killed in the middle of the cycle, the other process would be stuck waiting for data, but that is the tradeoff when using blocking I/O without polling.

[**Added**] The last test forks another process, so we have 3 in total. The parent process opens dm14-0 for writing, and the 2 other processes open dm14-1 for reading. Those processes will cycle for the rights to read the data input by the parent. If the mutex locks do not work, a process might read half an integer and the next would read the other half. If either process gets an unexpected result, it will output it.

**Output from the test program**

```
Non-blocking:
0: got 500500, expected 500500
1: got 510500, expected 510500

Blocking and forked:
0: expected 500500
0: got 500500
1: expected 1500500
1: got 1500500

No output from the multi-process test, as expected.
```

**Result:** Works as expected.

## The Conclusion

Building on the excellent foundation of `scull/pipe.c`, the implementation was simple and works as expected in all cases.

## Source Listing 1: The System Call Implementation

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif
#ifndef MODULE
#  define MODULE
#endif

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h>     /* everything... */
#include <linux/errno.h>  /* error codes */
#include <linux/types.h>  /* size_t */
#include <linux/fcntl.h>
#include <linux/poll.h>
#include <linux/unistd.h>
#include <linux/init.h>
#include <asm/system.h>
#include <asm/uaccess.h>

#ifndef min
#  define min(a,b) ((a)<(b) ? (a) : (b)) /* we use it in this file */
#endif

#define BUF_SIZE 4072
#define DEV_NAME "dm14_dev"

typedef struct DM14_Buffer {
    wait_queue_head_t inq, outq;    /* read and write queues */
    char *buffer, *end;             /* begin of buf, end of buf */
    char *rp, *wp;                  /* where to read, where to write */
    struct semaphore sem;           /* mutual exclusion semaphore */
} DM14_Buffer;
DM14_Buffer *devs;

int dm14_open(struct inode *inode, struct file *filp) {
    unsigned int num = MINOR(inode->i_rdev);

    if (filp->f_mode & FMODE_READ) {
        if (down_interruptible(&devs[num].sem))
            return -ERESTARTSYS;
        devs[num].rp = devs[num].buffer;
        up(&devs[num].sem);
    }

    num = (num+1)%2;
    if (filp->f_mode & FMODE_WRITE) {
        if (down_interruptible(&devs[num].sem))
            return -ERESTARTSYS;
        devs[num].wp = devs[num].buffer;
        up(&devs[num].sem);
    }

    if (filp->f_flags & (O_WRONLY | O_TRUNC)) {
        if (down_interruptible(&devs[num].sem))
            return -ERESTARTSYS;
        devs[num].rp = devs[num].wp = devs[num].buffer;
        memset(devs[num].buffer, 0, BUF_SIZE);
        up(&devs[num].sem);
    }

    return 0;
}

int dm14_release(struct inode *inode, struct file *filp) {
    return 0;
}

ssize_t dm14_read (struct file *filp, char *buf, size_t count, loff_t *f_pos) {
    unsigned int num = MINOR(filp->f_dentry->d_inode->i_rdev);

    if (f_pos != &filp->f_pos)
        return -ESPIPE;

    if (down_interruptible(&devs[num].sem))
        return -ERESTARTSYS;

    while (devs[num].rp == devs[num].wp) { /* nothing to read */
        up(&devs[num].sem); /* release the lock */
        if (filp->f_flags & O_NONBLOCK)
```

```
            return -EWOULDBLOCK;
        if (wait_event_interruptible(devs[num].inq, (devs[num].rp != devs[num].wp)))
            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        /* otherwise loop, but first reacquire the lock */
        if (down_interruptible(&devs[num].sem))
            return -ERESTARTSYS;
    }
    /* ok, data is there, return something */
    if (devs[num].wp > devs[num].rp)
        count = min(count, devs[num].wp - devs[num].rp);
    else /* the write pointer has wrapped, return data up to devs[num].end */
        count = min(count, devs[num].end - devs[num].rp);
    if (copy_to_user(buf, devs[num].rp, count)) {
        up (&devs[num].sem);
        return -EFAULT;
    }
    devs[num].rp += count;
    if (devs[num].rp == devs[num].end)
        devs[num].rp = devs[num].buffer; /* wrapped */
    up (&devs[num].sem);

    /* finally, awake any writers and return */
    wake_up_interruptible(&devs[num].outq);

    return count;
}

static inline int spacefree(int num) {
    if (devs[num].rp == devs[num].wp)
        return BUF_SIZE - 1;
    return ((devs[num].rp + BUF_SIZE - devs[num].wp) % BUF_SIZE) - 1;
}

ssize_t dm14_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos) {
    unsigned int num = (MINOR(filp->f_dentry->d_inode->i_rdev)+1)%2;

    if (f_pos != &filp->f_pos)
        return -ESPIPE;

    if (down_interruptible(&devs[num].sem))
        return -ERESTARTSYS;

    /* Make sure there's space to write */
    while (spacefree(num) == 0) { /* full */
        up(&devs[num].sem);
        if (filp->f_flags & O_NONBLOCK)
            return -EWOULDBLOCK;
        if (wait_event_interruptible(devs[num].outq, spacefree(num) > 0))
            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        if (down_interruptible(&devs[num].sem))
            return -ERESTARTSYS;
    }

    /* ok, space is there, accept something */
    count = min(count, spacefree(num));
    if (devs[num].wp >= devs[num].rp)
        count = min(count, devs[num].end - devs[num].wp); /* up to end-of-buffer */
    else /* the write pointer has wrapped, fill up to rp-1 */
        count = min(count, devs[num].rp - devs[num].wp - 1);
    if (copy_from_user(devs[num].wp, buf, count)) {
        up (&devs[num].sem);
        return -EFAULT;
    }
    devs[num].wp += count;
    if (devs[num].wp == devs[num].end)
        devs[num].wp = devs[num].buffer; /* wrapped */
    up(&devs[num].sem);

    /* finally, awake any reader */
    wake_up_interruptible(&devs[num].inq);  /* blocked in read() and select() */

    return count;
}

int dm14_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg) {
    return 0;
}

loff_t dm14_llseek(struct file *filp,  loff_t off, int whence) {
    return -ESPIPE; /* unseekable */
}

struct file_operations dm14_fops = {
llseek:     dm14_llseek,
```

```
read:       dm14_read,
write:      dm14_write,
ioctl:  dm14_ioctl,
open:       dm14_open,
release:    dm14_release,
};

int dm14_init(void) {
    int num=0;

    if ((num = register_chrdev(254, DEV_NAME, &dm14_fops)) < 0) {
        printk(KERN_WARNING "dm14: couldn't register device as 254\n");
        return num;
    }

    devs = kmalloc(2 * sizeof(DM14_Buffer), GFP_KERNEL);
    if (devs == NULL)
        return -ENOMEM;
    memset(devs, 0, 2 * sizeof(DM14_Buffer));

    for (num = 0; num < 2; num++) {
        devs[num].buffer = kmalloc(BUF_SIZE, GFP_KERNEL);
        if (!devs[num].buffer)
            return -ENOMEM;
        memset(devs[num].buffer, 0, BUF_SIZE);
        devs[num].rp = devs[num].wp = devs[num].buffer;
        devs[num].end = &devs[num].buffer[BUF_SIZE-1];
        init_waitqueue_head(&(devs[num].inq));
        init_waitqueue_head(&(devs[num].outq));
        sema_init(&devs[num].sem, 1);
    }

    printk("DM14: Init\n");

    return 0;
}

void dm14_cleanup(void) {
    int num=0;

    if ((num = unregister_chrdev(254, DEV_NAME)) < 0) {
        printk(KERN_WARNING "dm14: couldn't unregister device as 254\n");
    }

    if (!devs)
        return; /* nothing else to release */

    for (num=0; num < 2; num++) {
        if (devs[num].buffer)
            kfree(devs[num].buffer);
    }

    kfree(devs);
    devs = NULL;

    printk("DM14: Cleanup\n");
}

module_init(dm14_init);
module_exit(dm14_cleanup);
```

**Source Listing 2: The Test Program**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <linux/errno.h>
#include <stdlib.h>
#include <fcntl.h>

#define ITS 10000
extern int errno;

int main(int argc, char *argv[]) {
    pid_t pid=0;
    FILE *fd=NULL, *d[2] = {NULL, NULL};
    unsigned int sum=0, i=0, j=0, k[4] = {0,0,0,0}, val=0,
        sums[8] = {0,10,0,0,0,0,0,0};
    int err=0;

    d[0] = fopen("/dev/dm14-0", "rb+");
    d[1] = fopen("/dev/dm14-1", "rb+");
    for (i=0;i<2;i++) {
        if (!d[i]) {
            perror("D");
            return 0;
        }
        if ((val = fcntl(fileno(d[i]), F_SETFL, O_NONBLOCK)) < 0) {
            perror("F");
            return 0;
        }
    }

    while(k[0]<ITS || k[1]<ITS || k[2]<ITS || k[3]<ITS) {
        printf("Looping: %u %u %u %u", k[0], k[1], k[2], k[3]);

        for(j=0;j<2;j++) {
            if (k[j+2] < ITS) {
                errno = 0;
                sums[j]++;
                sums[j+2] += sums[j];
                fwrite(&sums[j], sizeof(int), 1, d[j]);
                err = ferror(d[j]);
                if ((err != 0) && (errno != EWOULDBLOCK) && (errno != 0)) {
                    printf("E: %i %i\n", err, errno);
                    perror("w");
                    return 0;
                } else if (errno != EWOULDBLOCK) { // write successful
                    k[j+2]++;
                } else { // would have blocked, so reset
                    sums[j+2] -= sums[j];
                    sums[j]--;
                }
                printf(" %i %i", err, errno);
            }

            if (k[j] < ITS) {
                errno = 0;
                fread(&sums[j+6], sizeof(int), 1, d[j]);
                err = ferror(d[j]);
                if ((err != 0) && (errno != EWOULDBLOCK) && (errno != 0)) {
                    printf("E: %i %i\n", err, errno);
                    perror("R");
                    return 0;
                } else if (errno != EWOULDBLOCK) { // success
                    sums[j+4] += sums[j+6];
                    k[j]++;
                }
```

```
                printf(" %i %i", err, errno);
            }
        }
        printf("\r");
    }

    printf("\r\n");
    printf("0: got %u, expected %u\n", sums[5], sums[2]);
    printf("1: got %u, expected %u\n", sums[4], sums[3]);

    pid = fork();

    if (pid == 0) {
        fd = fopen("/dev/dm14-0", "rb+");
        if (!fd) {
            perror("D0");
            return 0;
        }

        for (i=0; i<ITS; i++) {
            val++;
            sum += val;
            fwrite(&val, sizeof(int), 1, fd);
            if (ferror(fd)) {
                perror("W0");
                return 0;
            }
        }
        printf("0: expected %u\n", sum);

        sum = 0;

        for (i=0; i<ITS; i++) {
            fread(&val, sizeof(int), 1, fd);
            if (ferror(fd)) {
                perror("R0");
                return 0;
            }
            sum += val;
        }
        printf("1: got %u\n", sum);
    } else {
        fd = fopen("/dev/dm14-1", "rb+");
        if (!fd) {
            perror("D1");
            return 0;
        }

        for (i=0; i<ITS; i++) {
            fread(&val, sizeof(int), 1, fd);
            if (ferror(fd)) {
                perror("W1");
                return 0;
            }
            sum += val;
        }
        printf("0: got %u\n", sum);

        sum = 0;

        for (i=0; i<ITS; i++) {
            val++;
            sum += val;
            fwrite(&val, sizeof(int), 1, fd);
            if (ferror(fd)) {
                perror("R1");
                return 0;
```

```
            }
        }
        printf("1: expected %d\n", sum);
    }

    fclose(fd);

    // Get a third process into the game.
    if (pid == 0)
        pid = fork();

    if (pid == 0) {
        val = 5678;
        err = 8765;
        fd = fopen("/dev/dm14-0", "rb+");
        for(i=0 ; i<ITS/2 ; i++) {
            fwrite(&val, sizeof(int), 1, fd);
            if (ferror(fd)) {
                perror("W0");
                return 0;
            }
            fwrite(&err, sizeof(int), 1, fd);
            if (ferror(fd)) {
                perror("W1");
                return 0;
            }
        }
    } else {
        // this block of code will be run by 2 processes.
        fd = fopen("/dev/dm14-1", "rb+");
        for(i=0 ; i<ITS/2 ; i++) {
            fread(&val, sizeof(int), 1, fd);
            if (ferror(fd)) {
                perror("R");
                return 0;
            }
            if (val != 5678 && val != 8765)
                printf("%u got %i\n", pid, val);
        }
    }

    fclose(fd);

    return 0;
}
```